Memory Management

Contents of this section:

- Can I drop the `[]` when `delete`ing an array of some built-in type (`char`, `int`, etc)?
- After `p = new Fred[n]`, how does the compiler know there are `n` objects to be destructed during `delete[] p`?
- Is it legal (and moral) for a member function to say `delete this`?
- How do I allocate multidimensional arrays using `new`?
- But the previous FAQ's code is SOOOO tricky and error prone! Isn't there a simpler way?
- But the above `Matrix` class is specific to `Fred`! Isn't there a way to make it generic?
- What's another way to build a `Matrix` template?
- Does C++ have arrays whose length can be specified at run-time?
- How can I force objects of my class to always be created via `new` rather than as local, namespace-scope, global, or `static`?
- How do I do simple reference counting?
- How do I provide reference counting with copy-on-write semantics?
- How do I provide reference counting with copy-on-write semantics for a hierarchy of classes?
- Can I absolutely *prevent* people from subverting the reference counting mechanism, and if so, *should* I?
- Can I use a garbage collector in C++?
- What are the two kinds of garbage collectors for C++?
- Where can I get more info on garbage collectors for C++?
- What is an `auto_ptr` and why isn't there an `auto_array`?

FAQ How do I deal with memory leaks?

By writing code that doesn't have any. Clearly, if your code has `new` operations, `delete` operations, and pointer arithmetic all over the place, you are going to mess up somewhere and get leaks, stray pointers, etc. This is true independently of how conscientious you are with your allocations: eventually the complexity of the code will overcome the time and effort you can afford.

It follows that successful techniques rely on hiding allocation and deallocation inside more manageable types: For single objects, prefer `make_unique` or `make_shared`. For multiple objects, prefer using standard containers like `vector` and `unordered_map` as they manage memory for their elements better than you could without disproportionate effort. Consider writing this without the help of `string` and `vector`:

```
#include<vector>
#include<string>
#include<iostream>
#include<algorithm>
using namespace std;

int main()  // small program messing around with strings
{
    cout << "enter some whitespace-separated words:\n";
    vector<string> v;
    string s;
    while (cin>>s) v.push_back(s);

    sort(v.begin(),v.end());

    string cat;
    for (auto & str : v) cat += str+"+";
    cout << cat << '\n';
}
```

What would be your chance of getting it right the first time? And how would you know you didn't have a leak?

Note the absence of explicit memory management, macros, casts, overflow checks, explicit size limits, and pointers. By using a function object and a standard algorithm, the code could additionally have eliminated the pointer-like use of the iterator, but that seemed overkill for such a tiny program.

These techniques are not perfect and it is not always easy to use them systematically. However, they apply surprisingly widely and by reducing the number of explicit allocations and deallocations you make the remaining examples much easier to keep track of. As early as 1981, Stroustrup pointed out that by reducing the number of objects that he had to keep track of explicitly from many tens of thousands to a few dozens, he had reduced the intellectual effort needed to get the program right from a Herculean task to something manageable, or even easy.

If your application area doesn't have libraries that make programming that minimizes explicit memory management easy, then the fastest way of getting your program complete and correct might be to first build such a library.

Templates and the standard libraries make this use of containers, resource handles, etc., much easier than it was even a few years ago. The use of exceptions makes it close to essential.

If you cannot handle allocation/deallocation implicitly as part of an object you need in your application anyway, you can use a resource handle to minimize the chance of a leak. Here is an example where you need to return an object allocated on the free store from a function. This is an opportunity to forget to delete that object. After all, we cannot tell just looking at pointer whether it needs to be deallocated and if so who is responsible for that. Using a resource handle, here the standard library `unique_ptr`, makes it clear where the responsibility lies:

```
#include<memory>
#include<iostream>
using namespace std;

struct S {
    S() { cout << "make an S\n"; }
    ~S() { cout << "destroy an S\n"; }
    S(const S&) { cout << "copy initialize an S\n"; }
    S& operator=(const S&) { cout << "copy assign an S\n"; }
};

S* f()
{
    return new S;   // who is responsible for deleting this S?
};

unique_ptr<S> g()
{
    return make_unique<S>();    // explicitly transfer responsibili
}

int main()
{
    cout << "start main\n";
    S* p = f();
    cout << "after f() before g()\n";
//  S* q = g(); // this error would be caught by the compiler
    unique_ptr<S> q = g();
    cout << "exit main\n";
    // leaks *p
    // implicitly deletes *q
}
```

Think about resources in general, rather than simply about memory.

If systematic application of these techniques is not possible in your environment (you have to use code from elsewhere, part of your program was written by Neanderthals, etc.), be sure to use a memory leak detector as part of your standard development procedure, or plug in a garbage collector.

FAQ Can I use `new` just as in Java?

Sort of, but don't do it blindly, if you do want it prefer to spell it as `make_unique` or `make_shared`, and there are often superior alternatives that are simpler and more robust than any of that. Consider:

```
void compute(cmplx z, double d)
{
    cmplx z2 = z+d; // c++ style
    z2 = f(z2);     // use z2

    cmplx& z3 = *new cmplx(z+d);    // Java style (assuming Java could
    z3 = f(z3);
    delete &z3;
}
```

The clumsy use of `new` for `z3` is unnecessary and slow compared with the idiomatic use of a local variable (`z2`). You don't need to use `new` to create an object if you also `delete` that object in the same scope; such an object should be a local variable.

FAQ Should I use `NULL` or `0` or `nullptr`?

You should use `nullptr` as the null pointer value. The others still work for backward compatibility with older code.

A problem with both `NULL` and `0` as a null pointer value is that `0` is a special "maybe an integer value and maybe a pointer" value. Use `0` only for integers, and that confusion disappears.

FAQ Does `delete p` delete the pointer p, or the pointed-to-data *p?

The pointed-to-data.

The keyword should really be `delete_the_thing_pointed_to_by`. The same abuse of English occurs when `free`ing the memory pointed to by a pointer in C: `free(p)` really means `free_the_stuff_pointed_to_by(p)`.

FAQ Is it safe to `delete` the same pointer twice?

No! (Assuming you didn't get that pointer back from `new` in between.)

For example, the following is a disaster:

```
class Foo { /*...*/ };

void yourCode()
{
  Foo* p = new Foo();
  delete p;
  delete p;  // DISASTER!
  // ...
}
```

That second `delete p` line might do some really bad things to you. It might, depending on the phase of the moon, corrupt your heap, crash your program, make arbitrary and bizarre changes to objects that are already out there on the heap, etc. Unfortunately these symptoms can appear and disappear randomly. According to Murphy's law, you'll be hit the hardest at the worst possible moment (when the customer is looking, when a high-value transaction is trying to post, etc.).

Note: *some* runtime systems will protect you from certain very simple cases of double `delete`. Depending on the details, you might be okay *if* you happen to be running on one of those systems *and if* no one ever deploys your code on another system that handles

things differently *and if* you are deleting something that doesn't have a destructor *and if* you don't do anything significant between the two `delete`s *and if* no one ever changes your code to do something significant between the two `delete`s *and if* your thread scheduler (over which you likely have no control!) doesn't happen to swap threads between the two `delete`s *and if*, *and if*, *and if*. So back to Murphy: since it can go wrong, it will, and it will go wrong at the worst possible moment.

Do *NOT* email me saying you tested it and it doesn't crash. Get a clue. A non-crash doesn't prove the absence of a bug; it merely fails to prove the presence of a bug.

Trust me: double-`delete` is bad, bad, bad. Just say no.

FAQ Can I `free()` pointers allocated with `new`? Can I `delete` pointers allocated with `malloc()`?

No! In brief, conceptually `malloc` and `new` allocate from different heaps, so can't `free` or `delete` each other's memory. They also operate at different levels – raw memory vs. constructed objects.

You can use `malloc()` and `new` in the same program. But you cannot allocate an object with `malloc()` and free it using `delete`. Nor can you allocate with `new` and `delete` with `free()` or use `realloc()` on an array allocated by `new`.

The C++ operators `new` and `delete` guarantee proper construction and destruction; where constructors or destructors need to be invoked, they are. The C-style functions `malloc()`, `calloc()`, `free()`, and `realloc()` don't ensure that. Furthermore, there is no guarantee that the mechanism used by `new` and `delete` to acquire and release raw memory is compatible with `malloc()` and `free()`. If mixing styles works on your system, you were simply "lucky" – for now.

If you feel the need for `realloc()` – and many do – then consider using a standard library `vector`. For example

```
// read words from input into a vector of strings:

vector<string> words;
string s;
while (cin>>s && s!=".") words.push_back(s);
```

The `vector` expands as needed.

See also the examples and discussion in "Learning Standard C++ as a New Language", which you can download from Stroustrup's publications list .

FAQ What is the difference between `new` and `malloc()`?

First, `make_unique` (or `make_shared`) are nearly always superior to both `new` and `malloc()` and completely eliminate `delete` and `free()`.

Having said that, here's the difference between those two:

`malloc()` is a function that takes a number (of bytes) as its argument; it returns a `void*` pointing to unitialized storage. `new` is an operator that takes a type and (optionally) a set of initializers for that type as its arguments; it returns a pointer to an (optionally) initialized object of its type. The difference is most obvious when you want to allocate an object of a user-defined type with non-trivial initialization semantics. Examples:

```
class Circle : public Shape {
public:
    Circle(Point c, int r);
    // no default constructor
    // ...
};

class X {
public:
    X();    // default constructor
    // ...
};

void f(int n)
{
```

```
    void* p1 = malloc(40);  // allocate 40 (uninitialized) bytes

    int* p2 = new int[10];  // allocate 10 uninitialized ints
    int* p3 = new int(10);  // allocate 1 int initialized to 10
    int* p4 = new int();    // allocate 1 int initialized to 0
    int* p4 = new int;  // allocate 1 uninitialized int

    Circle* pc1 = new Circle(Point(0,0),10); // allocate a Circle c
                              // with the specified argument
    Circle* pc2 = new Circle;   // error no default constructor

    X* px1 = new X;     // allocate a default constructed X
    X* px2 = new X();   // allocate a default constructed X
    X* px2 = new X[10]; // allocate 10 default constructed Xs
    // ...
}
```

Note that when you specify a initializer using the "(value)" notation, you get initialization with that value. Unfortunately, you cannot specify that for an array. Often, a `vector` is a better alternative to a free-store-allocated array (e.g., consider exception safety).

Whenever you use `malloc()` you must consider initialization and conversion of the return pointer to a proper type. You will also have to consider if you got the number of bytes right for your use. There is no performance difference between `malloc()` and `new` when you take initialization into account.

`malloc()` reports memory exhaustion by returning `0`. `new` reports allocation and initialization errors by throwing exceptions (`bad_alloc`).

Objects created by `new` are destroyed by `delete`. Areas of memory allocated by `malloc()` are deallocated by `free()`.

FAQ Why should I use new instead of trustworthy old malloc()?

First, `make_unique` (or `make_shared`) are nearly always superior to both `new` and `malloc()` and completely eliminate `delete` and `free()`.

Having said that, benefits of using `new` instead of `malloc` are: Constructors/destructors, type safety, overridability.

- Constructors/destructors: unlike `malloc(sizeof(Fred))`, `new Fred()` calls `Fred`'s constructor. Similarly, `delete p` calls `*p`'s destructor.
- Type safety: `malloc()` returns a `void*` which isn't type safe. `new Fred()` returns a pointer of the right type (a `Fred*`).
- Overridability: `new` is an `operator` that can be overridden by a class, while `malloc()` is not overridable on a per-class basis.

FAQ Can I use `realloc()` on pointers allocated via `new`?

No!

When `realloc()` has to copy the allocation, it uses a *bitwise* copy operation, which will tear many C++ objects to shreds. C++ objects should be allowed to copy themselves. They use their own copy constructor or assignment operator.

Besides all that, the heap that `new` uses may *not* be the same as the heap that `malloc()` and `realloc()` use!

FAQ Why doesn't C++ have an equivalent to `realloc()`?

If you want to, you can of course use `realloc()`. However, `realloc()` is only guaranteed to work on arrays allocated by `malloc()` (and similar functions) containing objects without user-defined copy constructors. Also, please remember that contrary to naive expectations, `realloc()` occasionally does copy its argument array.

In C++, a better way of dealing with reallocation is to use a standard library container, such as `vector`, and let it grow naturally.

FAQ Do I need to check for null after `p = new Fred()`?

No! (But if you have an ancient, stone-age compiler, you may have to force the `new` operator to `throw` an exception if it runs out of memory.)

It turns out to be a real pain to always write explicit `nullptr` tests after every `new` allocation. Code like the following is very tedious:

```
Fred* p = new Fred();
if (nullptr == p)    // Only needed if your compiler is from the Stone
  throw std::bad_alloc();
```

If your compiler doesn't support (or if you refuse to use) exceptions, your code might be even more tedious:

```
Fred* p = new Fred();
if (nullptr == p) {    // Only needed if your compiler is from the Ston
  std::cerr << "Couldn't allocate memory for a Fred" << std::endl;
  abort();
}
```

Take heart. In C++, if the runtime system cannot allocate `sizeof(Fred)` bytes of memory during `p = new Fred()`, a `std::bad_alloc` exception will be thrown. Unlike `malloc()`, `new` *never* returns null!

Therefore you should simply write:

```
Fred * p = new Fred();  // No need to check if p is null
```

On the second thought. Scratch that. You should simply write:

```
auto p = make_unique<Fred>();  // No need to check if p is null
```

There, there... Much better now!

However, if your compiler is ancient, it may not yet support this. Find out by checking your compiler's documentation under "`new`". If it is ancient, you may have to force the compiler to have this behavior.

FAQ How can I convince my (older) compiler to automatically check new to see if it returns null?

Eventually your compiler will.

If you have an old compiler that doesn't automagically perform the null test, you can force the runtime system to do the test by installing a "new handler" function. Your "new handler" function can do anything you want, such as `throw` an exception, `delete` some objects and return (in which case `operator new` will retry the allocation), print a message and `abort()` the program, etc.

Here's a sample "new handler" that prints a message and `throw`s an exception. The handler is installed using `std::set_new_handler()`:

```
#include <new>       // To get std::set_new_handler
#include <cstdlib>   // To get abort()
#include <iostream>  // To get std::cerr

class alloc_error : public std::exception {
public:
  alloc_error() : exception() { }
};

void myNewHandler()
{
  // This is your own handler.  It can do anything you want.
  throw alloc_error();
}

int main()
{
  std::set_new_handler(myNewHandler);   // Install your "new handler"
  // ...
}
```

After the `std::set_new_handler()` line is executed, `operator new` will call your `myNewHandler()` if/when it runs out of memory. This means that `new` will never return null:

```
Fred* p = new Fred();   // No need to check if p is null
```

Note: If your compiler doesn't support exception handling, you can, as a last resort, change the line `throw ...;` to:

```
  std::cerr << "Attempt to allocate memory failed!" << std::endl;
  abort();
```

Note: If some namespace-scope / global / static object's constructor uses `new`, it might not use the `myNewHandler()` function since that constructor often gets called before `main()` begins. Unfortunately there's no convenient way to guarantee that the `std::set_new_handler()` will be called before the first use of `new`. For example, even if you put the `std::set_new_handler()` call in the constructor of a global object, you still don't know if the module ("compilation unit") that contains that global object will be elaborated first or last or somewhere inbetween. Therefore you still don't have any guarantee that your call of `std::set_new_handler()` will happen before any other namespace-scope / global's constructor gets invoked.

FAQ Do I need to check for null before `delete p`?

No!

The C++ language guarantees that `delete p` will do nothing if `p` is null. Since you might get the test backwards, and since most testing methodologies force you to explicitly test every branch point, you should *not* put in the redundant `if` test.

Wrong:

```
if (p != nullptr)   // or just "if (p)"
  delete p;
```

Right:

```
delete p;
```

FAQ What are the two steps that happen when I say `delete p`?

`delete p` is a two-step process: it calls the destructor, then releases the memory. The code generated for `delete p` is functionally similar to this (assuming `p` is of type `Fred*`):

```
// Original code: delete p;
if (p) {    // or "if (p != nullptr)"
  p->~Fred();
  operator delete(p);
}
```

The statement `p->~Fred()` calls the destructor for the `Fred` object pointed to by `p`.

The statement `operator delete(p)` calls the memory deallocation primitive, `void operator delete(void* p)`. This primitive is similar in spirit to `free(void* p)`. (Note, however, that these two are *not* interchangeable; e.g., there is no guarantee that the two memory deallocation primitives even use the same heap!)

FAQ Why doesn't `delete` null out its operand?

First, you should normally be using smart pointers, so you won't care – you won't be writing `delete` anyway.

For those rare cases where you really are doing manual memory management and so do care, consider:

```
delete p;
// ...
delete p;
```

If the `...` part doesn't touch `p` then the second `delete p;` is a serious error that a C++ implementation cannot effectively protect itself against (without unusual precautions). Since deleting a null pointer is harmless by definition, a simple solution would be for `delete p;` to do a `p=nullptr;` after it has done whatever else is required. However, C++ doesn't guarantee that.

One reason is that the operand of `delete` need not be an lvalue. Consider:

```
delete p+1;
delete f(x);
```

Here, the implementation of `delete` does not have a pointer to which it can null out. These examples may be rare, but they do imply that it is not possible to guarantee that "any pointer to a deleted object is null." A simpler way of bypassing that "rule" is to have two pointers to an object:

```
T* p = new T;
T* q = p;
delete p;
delete q;   // ouch!
```

C++ explicitly allows an implementation of `delete` to null out an lvalue operand, but that idea doesn't seem to have become popular with implementers.

If you consider zeroing out pointers important, consider using a destroy function:

```
template<class T> inline void destroy(T*& p) { delete p; p = 0; }
```

Consider this yet-another reason to minimize explicit use of `new` and `delete` by relying on standard library smart pointers, containers, handles, etc.

Note that passing the pointer as a reference (to allow the pointer to be nulled out) has the added benefit of preventing `destroy()` from being called for an rvalue:

```
int* f();
int* p;
// ...
destroy(f());   // error: trying to pass an rvalue by non-const ref
destroy(p+1);   // error: trying to pass an rvalue by non-const ref
```

FAQ Why isn't the destructor called at the end of scope?

The simple answer is "of course it is!", but have a look at the kind of example that often accompany that question:

```
void f()
{
    X* p = new X;
    // use p
}
```

That is, there was some (mistaken) assumption that the object created by `new` would be destroyed at the end of a function.

Basically, you should only use heap allocation if you want an object to live beyond the lifetime of the scope you create it in. Even then, you should normally use `make_unique` or `make_shared`. In those rare cases where you do want heap allocation and you opt to use `new`, you need to use `delete` to destroy the object. For example:

```
X* g(int i) { /* ... */ return new X(i); }  // the X outlives the c

void h(int i)
{
    X* p = g(i);
    // ...
    delete p; // caveat: not exception safe
}
```

If you want an object to live in a scope only, don't use heap allocation at all but simply define a variable:

```
{
        ClassName x;
        // use x
}
```

The variable is implicitly destroyed at the end of the scope.

Code that creates an object using `new` and then `delete`s it at the end of the same scope is ugly, error-prone, inefficient, and usually not exception-safe. For example:

```
void very_bad_func()    // ugly, error-prone, and inefficient
{
    X* p = new X;
    // use p
    delete p;  // not exception-safe
}
```

FAQ In `p = new Fred()`, does the `Fred` memory "leak" if the `Fred` constructor throws an exception?

No.

If an exception occurs during the `Fred` constructor of `p = new Fred()`, the C++ language guarantees that the memory `sizeof(Fred)` bytes that were allocated will automagically be released back to the heap.

Here are the details: `new Fred()` is a two-step process:

1. `sizeof(Fred)` bytes of memory are allocated using the primitive `void* operator new(size_t nbytes)`. This primitive is similar in spirit to `malloc(size_t nbytes)`. (Note, however, that these two are *not* interchangeable; e.g., there is no guarantee that the two memory allocation primitives even use the same heap!).
2. It constructs an object in that memory by calling the `Fred` constructor. The pointer returned from the first step is passed as the `this` parameter to the constructor. This step is wrapped in a `try` ... `catch` block to handle the case when an exception is thrown during this step.

Thus the actual generated code is functionally similar to:

```
// Original code: Fred* p = new Fred();
Fred* p;
void* tmp = operator new(sizeof(Fred));
try {
  new(tmp) Fred();  // Placement new
  p = (Fred*)tmp;   // The pointer is assigned only if the ctor succeea
}
catch (...) {
  operator delete(tmp);  // Deallocate the memory
  throw;                 // Re-throw the exception
}
```

The statement marked "Placement `new`" calls the `Fred` constructor. The pointer `p` becomes the `this` pointer inside the constructor, `Fred::Fred()`.

FAQ How do I allocate / unallocate an array of things?

Use `p = new T[n]` and `delete[] p`:

```
Fred* p = new Fred[100];
// ...
delete[] p;
```

Any time you allocate an array of objects via `new` (usually with the `[n]` in the `new` expression), you *must* use `[]` in the `delete` statement. This syntax is necessary because there is no syntactic difference between a pointer to a thing and a pointer to an array of things (something we inherited from C).

FAQ What if I forget the `[]` when `deleteing` an array allocated via `new T[n]`?

All life comes to a catastrophic end.

It is the programmer's —not the compiler's— responsibility to get the connection between `new T[n]` and `delete[] p` correct. If you get it wrong, neither a compile-time nor a run-time error message will be generated by the compiler. Heap corruption is a likely result. Or worse. Your program will probably die.

FAQ Can I drop the `[]` when `deleteing` an array of some built-in type (`char`, `int`, etc)?

No!

Sometimes programmers think that the `[]` in the `delete[] p` only exists so the compiler will call the appropriate destructors for all

elements in the array. Because of this reasoning, they assume that an array of some built-in type such as `char` or `int` can be `delete`d without the `[]`. E.g., they assume the following is valid code:

```
void userCode(int n)
{
  char* p = new char[n];
  // ...
  delete p;     // ← ERROR! Should be delete[] p !
}
```

But the above code is wrong, and it can cause a disaster at runtime. In particular, the code that's called for `delete p` is `operator delete(void*)`, but the code that's called for `delete[] p` is `operator delete[](void*)`. The default behavior for the latter is to call the former, but users are allowed to replace the latter with a different behavior (in which case they would normally also replace the corresponding `new` code in `operator new[](size_t)`). If they replaced the `delete[]` code so it wasn't compatible with the `delete` code, and you called the wrong one (i.e., if you said `delete p` rather than `delete[] p`), you could end up with a disaster at runtime.

FAQ After `p = new Fred[n]`, how does the compiler know there are `n` objects to be destructed during `delete[] p`?

Short answer: Magic.

Long answer: The run-time system stores the number of objects, `n`, somewhere where it can be retrieved if you only know the pointer, `p`. There are two popular techniques that do this. Both these techniques are in use by commercial-grade compilers, both have tradeoffs, and neither is perfect. These techniques are:

- Over-allocate the array and put `n` just to the left of the first `Fred` object.
- Use an associative array with `p` as the key and `n` as the value.

FAQ Is it legal (and moral) for a member function to say `delete this`?

As long as you're careful, it's okay (not evil) for an object to commit suicide (`delete this`).

Here's how I define "careful":

1. You must be absolutely 100% positively sure that `this` object was allocated via `new` (not by `new[]`, nor by placement `new`, nor a local object on the stack, nor a namespace-scope / global, nor a member of another object; but by plain ordinary `new`).
2. You must be absolutely 100% positively sure that your member function will be the last member function invoked on `this` object.
3. You must be absolutely 100% positively sure that the rest of your member function (after the `delete this` line) doesn't touch any piece of `this` object (including calling any other member functions or touching any data members). This includes code that will run in destructors for any objects allocated on the stack that are still alive.
4. You must be absolutely 100% positively sure that no one even touches the `this` pointer itself after the `delete this` line. In other words, you must not examine it, compare it with another pointer, compare it with `nullptr`, print it, cast it, do anything with it.

Naturally the usual caveats apply in cases where your `this` pointer is a pointer to a base class when you don't have a virtual destructor.

FAQ How do I allocate multidimensional arrays using new?

There are many ways to do this, depending on how flexible you want the array sizing to be. On one extreme, if you know all the dimensions at compile-time, you can allocate multidimensional arrays statically (as in C):

```
class Fred { /*...*/ };
void someFunction(Fred& fred);

void manipulateArray()
{
  const unsigned nrows = 10;  // Num rows is a compile-time constant
  const unsigned ncols = 20;  // Num columns is a compile-time constant
  Fred matrix[nrows][ncols];

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // Here's the way you access the (i,j) element:
      someFunction( matrix[i][j] );

      // You can safely "return" without any special delete code:
      if (today == "Tuesday" && moon.isFull())
        return;     // Quit early on Tuesdays when the moon is full
    }
  }

  // No explicit delete code at the end of the function either
}
```

More commonly, the size of the matrix isn't known until run-time but you know that it will be rectangular. In this case you need to use the heap ("freestore"), but at least you are able to allocate all the elements in one freestore chunk.

```
void manipulateArray(unsigned nrows, unsigned ncols)
{
  Fred* matrix = new Fred[nrows * ncols];

  // Since we used a simple pointer above, we need to be VERY
  // careful to avoid skipping over the delete code.
  // That's why we catch all exceptions:
  try {

    // Here's how to access the (i,j) element:
    for (unsigned i = 0; i < nrows; ++i) {
      for (unsigned j = 0; j < ncols; ++j) {
        someFunction( matrix[i*ncols + j] );
      }
    }

    // If you want to quit early on Tuesdays when the moon is full,
    // make sure to do the delete along ALL return paths:
    if (today == "Tuesday" && moon.isFull()) {
      delete[] matrix;
      return;
    }

    // ...code that fiddles with the matrix...

  }
  catch (...) {
    // Make sure to do the delete when an exception is thrown:
    delete[] matrix;
    throw;    // Re-throw the current exception
  }

  // Make sure to do the delete at the end of the function too:
  delete[] matrix;
}
```

Finally at the other extreme, you may not even be guaranteed that the matrix is rectangular. For example, if each row could have a different length, you'll need to allocate each row individually. In the following function, `ncols[i]` is the number of columns in row number `i`, where `i` varies between `0` and `nrows-1` inclusive.

```
void manipulateArray(unsigned nrows, unsigned ncols[])
{
  typedef Fred* FredPtr;

  // There will not be a leak if the following throws an exception:
  FredPtr* matrix = new FredPtr[nrows];

  // Set each element to null in case there is an exception later.
  // (See comments at the top of the try block for rationale.)
  for (unsigned i = 0; i < nrows; ++i)
    matrix[i] = nullptr;

  // Since we used a simple pointer above, we need to be
  // VERY careful to avoid skipping over the delete code.
  // That's why we catch all exceptions:
  try {

    // Next we populate the array.  If one of these throws, all
    // the allocated elements will be deleted (see catch below).
    for (unsigned i = 0; i < nrows; ++i)
      matrix[i] = new Fred[ ncols[i] ];

    // Here's how to access the (i,j) element:
    for (unsigned i = 0; i < nrows; ++i) {
      for (unsigned j = 0; j < ncols[i]; ++j) {
        someFunction( matrix[i][j] );
      }
    }

    // If you want to quit early on Tuesdays when the moon is full,
    // make sure to do the delete along ALL return paths:
    if (today == "Tuesday" && moon.isFull()) {
      for (unsigned i = nrows; i > 0; --i)
        delete[] matrix[i-1];
      delete[] matrix;
      return;
    }

    // ...code that fiddles with the matrix...

  }
  catch (...) {
    // Make sure to do the delete when an exception is thrown:
    // Note that some of these matrix[...] pointers might be
    // null, but that's okay since it's legal to delete null.
    for (unsigned i = nrows; i > 0; --i)
      delete[] matrix[i-1];
    delete[] matrix;
    throw;    // Re-throw the current exception
  }

  // Make sure to do the delete at the end of the function too.
  // Note that deletion is the opposite order of allocation:
  for (unsigned i = nrows; i > 0; --i)
    delete[] matrix[i-1];
  delete[] matrix;
}
```

Note the funny use of `matrix[i-1]` in the deletion process. This prevents wrap-around of the `unsigned` value when `i` goes one step below zero.

Finally, note that pointers and arrays are evil. It is normally much better to encapsulate your pointers in a class that has a safe and simple interface. The following FAQ shows how to do this.

FAQ But the previous FAQ's code is SOOOO tricky and error prone! Isn't there a simpler way?

Yep.

The reason the code in the previous FAQ was so tricky and error prone was that it used pointers, and we know that pointers and arrays are evil. The solution is to encapsulate your pointers in a class that has a safe and simple interface. For example, we can define a `Matrix` class that handles a rectangular matrix so our user code will be vastly simplified when compared to the the rectangular matrix code from the previous FAQ:

```
// The code for class Matrix is shown below...
void someFunction(Fred& fred);

void manipulateArray(unsigned nrows, unsigned ncols)
{
  Matrix matrix(nrows, ncols);   // Construct a Matrix called matrix

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // Here's the way you access the (i,j) element:
      someFunction( matrix(i,j) );

      // You can safely "return" without any special delete code:
      if (today == "Tuesday" && moon.isFull())
        return;     // Quit early on Tuesdays when the moon is full
    }
  }

  // No explicit delete code at the end of the function either
}
```

The main thing to notice is the lack of clean-up code. For example, there aren't any `delete` statements in the above code, yet there will

be *no* memory leaks, assuming only that the `Matrix` destructor does its job correctly.

Here's the `Matrix` code that makes the above possible:

```
class Matrix {
public:
  Matrix(unsigned nrows, unsigned ncols);
  // Throws a BadSize object if either size is zero
  class BadSize { };

  // Based on the Law Of The Big Three:
 ~Matrix();
  Matrix(const Matrix& m);
  Matrix& operator= (const Matrix& m);

  // Access methods to get the (i,j) element:
  Fred&       operator() (unsigned i, unsigned j);        // Subscript
  const Fred& operator() (unsigned i, unsigned j) const;  // Subscript
  // These throw a BoundsViolation object if i or j is too big
  class BoundsViolation { };

private:
  unsigned nrows_, ncols_;
  Fred* data_;
};

inline Fred& Matrix::operator() (unsigned row, unsigned col)
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

inline const Fred& Matrix::operator() (unsigned row, unsigned col) cons
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

Matrix::Matrix(unsigned nrows, unsigned ncols)
  : nrows_ (nrows)
  , ncols_ (ncols)
//, data_  ← initialized below after the if...throw statement
{
  if (nrows == 0 || ncols == 0)
    throw BadSize();
  data_ = new Fred[nrows * ncols];
}

Matrix::~Matrix()
{
  delete[] data_;
}
```

Note that the above `Matrix` class accomplishes two things: it moves some tricky memory management code from the user code (e.g., `main()`) to the class, and it reduces the overall bulk of program. The latter point is important. For example, assuming `Matrix` is even mildly reusable, moving complexity from the users [plural] of

`Matrix` into `Matrix` itself [singular] is equivalent to moving complexity from the many to the few. Anyone who has seen *Star Trek 2* knows that the good of the many outweighs the good of the few... or the one.

FAQ But the above `Matrix` class is specific to `Fred`! Isn't there a way to make it generic?

Yep; just use templates:

Here's how this can be used:

```
#include "Fred.h"  // To get the definition for class Fred

// The code for Matrix<T> is shown below...
void someFunction(Fred& fred);

void manipulateArray(unsigned nrows, unsigned ncols)
{
  Matrix<Fred> matrix(nrows, ncols);   // Construct a Matrix<Fred> call

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // Here's the way you access the (i,j) element:
      someFunction( matrix(i,j) );

      // You can safely "return" without any special delete code:
      if (today == "Tuesday" && moon.isFull())
        return;     // Quit early on Tuesdays when the moon is full
    }
  }

  // No explicit delete code at the end of the function either
}
```

Now it's easy to use `Matrix<T>` for things other than `Fred`. For example, the following uses a `Matrix` of `std::string` (where `std::string` is the standard string class):

```
#include <string>

void someFunction(std::string& s);

void manipulateArray(unsigned nrows, unsigned ncols)
{
  Matrix<std::string> matrix(nrows, ncols);   // Construct a Matrix<sta

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // Here's the way you access the (i,j) element:
      someFunction( matrix(i,j) );

      // You can safely "return" without any special delete code:
      if (today == "Tuesday" && moon.isFull())
```

```
      return;     // Quit early on Tuesdays when the moon is full
    }
  }

  // No explicit delete code at the end of the function either
}
```

You can thus get an entire *family* of classes from a template. For example, `Matrix<Fred>`, `Matrix<std::string>`, `Matrix<Matrix<std::string>>`, etc.

Here's one way that the template can be implemented:

```
template<typename T>  // See section on templates for more
class Matrix {
public:
  Matrix(unsigned nrows, unsigned ncols);
  // Throws a BadSize object if either size is zero
  class BadSize { };

  // Based on the Law Of The Big Three:
 ~Matrix();
  Matrix(const Matrix<T>& m);
  Matrix<T>& operator= (const Matrix<T>& m);

  // Access methods to get the (i,j) element:
  T&       operator() (unsigned i, unsigned j);        // Subscript ope
  const T& operator() (unsigned i, unsigned j) const;  // Subscript ope
  // These throw a BoundsViolation object if i or j is too big
  class BoundsViolation { };

private:
  unsigned nrows_, ncols_;
  T* data_;
};

template<typename T>
inline T& Matrix<T>::operator() (unsigned row, unsigned col)
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

template<typename T>
inline const T& Matrix<T>::operator() (unsigned row, unsigned col) cons
{
  if (row >= nrows_ || col >= ncols_)
    throw BoundsViolation();
  return data_[row*ncols_ + col];
}

template<typename T>
inline Matrix<T>::Matrix(unsigned nrows, unsigned ncols)
  : nrows_ (nrows)
  , ncols_ (ncols)
//, data_  ← initialized below after the if...throw statement
{
  if (nrows == 0 || ncols == 0)
    throw BadSize();
  data_ = new T[nrows * ncols];
}

template<typename T>
```

```
inline Matrix<T>::~Matrix()
{
  delete[] data_;
}
```

FAQ What's another way to build a `Matrix` template?

Use the standard `vector` template, and make a `vector` of `vector`.

The following uses a `std::vector<std::vector<T>>`.

```
#include <vector>

template<typename T>  // See section on templates for more
class Matrix {
public:
  Matrix(unsigned nrows, unsigned ncols);
  // Throws a BadSize object if either size is zero
  class BadSize { };

  // No need for any of The Big Three!

  // Access methods to get the (i,j) element:
  T&       operator() (unsigned i, unsigned j);        // Subscript ope
  const T& operator() (unsigned i, unsigned j) const;  // Subscript ope
  // These throw a BoundsViolation object if i or j is too big
  class BoundsViolation { };

  unsigned nrows() const;  // #rows in this matrix
  unsigned ncols() const;  // #columns in this matrix

private:
  std::vector<std::vector<T>> data_;
};

template<typename T>
inline unsigned Matrix<T>::nrows() const
{ return data_.size(); }

template<typename T>
inline unsigned Matrix<T>::ncols() const
{ return data_[0].size(); }

template<typename T>
inline T& Matrix<T>::operator() (unsigned row, unsigned col)
{
  if (row >= nrows() || col >= ncols()) throw BoundsViolation();
  return data_[row][col];
}

template<typename T>
inline const T& Matrix<T>::operator() (unsigned row, unsigned col) cons
{
  if (row >= nrows() || col >= ncols()) throw BoundsViolation();
  return data_[row][col];
}

template<typename T>
Matrix<T>::Matrix(unsigned nrows, unsigned ncols)
  : data_ (nrows)
{
```

```
  if (nrows == 0 || ncols == 0)
    throw BadSize();
  for (unsigned i = 0; i < nrows; ++i)
    data_[i].resize(ncols);
}
```

Note how much simpler this is than the previous: there is no explicit `new` in the constructor, and there is no need for any of The Big Three (destructor, copy constructor or assignment operator). Simply put, your code is a *lot* less likely to have memory leaks if you use `std::vector` than if you use explicit `new T[n]` and `delete[] p`.

Note also that `std::vector` doesn't force you to allocate numerous chunks of memory. If you prefer to allocate only one chunk of memory for the entire matrix, as was done in the previous, just change the type of `data_` to `std::vector<T>` and add member variables `nrows_` and `ncols_`. You'll figure out the rest: initialize `data_` using `data_(nrows * ncols)`, change `operator()()` to `return data_[row*ncols_ + col];`, etc.

FAQ Does C++ have arrays whose length can be specified at run-time?

Yes, in the sense that the standard library has a `std::vector` template that provides this behavior.

No, in the sense that built-in array types need to have their length specified at compile time.

Yes, in the sense that even built-in array types can specify the first index bounds at run-time. E.g., comparing with the previous FAQ, if you only need the first array dimension to vary then you can just ask new for an array of arrays, rather than an array of pointers to arrays:

```
const unsigned ncols = 100;          // ncols = number of columns in t
class Fred { /*...*/ };
void manipulateArray(unsigned nrows)  // nrows = number of rows in the
```

```
{
  Fred (*matrix)[ncols] = new Fred[nrows][ncols];
  // ...
  delete[] matrix;
}
```

You can't do this if you need anything other than the first dimension of the array to change at run-time.

But please, don't use arrays unless you have to. Arrays are evil. Use some object of some class if you can. Use arrays only when you have to.

FAQ How can I force objects of my class to always be created via new rather than as local, namespace-scope, global, or static?

Use the Named Constructor Idiom.

As usual with the Named Constructor Idiom, the constructors are all `private` or `protected`, and there are one or more `public static create()` methods (the so-called "named constructors"), one per constructor. In this case the `create()` methods allocate the objects via `new`. Since the constructors themselves are not `public`, there is no other way to create objects of the class.

```
class Fred {
public:
  // The create() methods are the "named constructors":
  static Fred* create()                 { return new Fred();     }
  static Fred* create(int i)            { return new Fred(i);    }
  static Fred* create(const Fred& fred) { return new Fred(fred); }
  // ...

private:
  // The constructors themselves are private or protected:
  Fred();
  Fred(int i);
  Fred(const Fred& fred);
  // ...
};
```

Now the only way to create `Fred` objects is via `Fred::create()`:

```
int main()
{
  Fred* p = Fred::create(5);
```

```
  // ...
  delete p;
  // ...
}
```

Make sure your constructors are in the `protected` section if you expect `Fred` to have derived classes.

Note also that you can make another class `Wilma` a `friend` of `Fred` if you want to allow a `Wilma` to have a member object of class `Fred`, but of course this is a softening of the original goal, namely to force `Fred` objects to be allocated via `new`.

FAQ How do I do simple reference counting?

If all you want is the ability to pass around a bunch of pointers to the same object, with the feature that the object will automagically get `deleted` when the last pointer to it disappears, you can use something like the following "smart pointer" class:

```
// Fred.h

class FredPtr;

class Fred {
public:
  Fred() : count_(0) /*...*/ { }  // All ctors set count_ to 0 !
  // ...
private:
  friend class FredPtr;     // A friend class
  unsigned count_;
  // count_ must be initialized to 0 by all constructors
  // count_ is the number of FredPtr objects that point at this
};

class FredPtr {
public:
  Fred* operator-> () { return p_; }
  Fred& operator* ()  { return *p_; }
  FredPtr(Fred* p)    : p_(p) { ++p_->count_; }  // p must not be null
 ~FredPtr()           { if (--p_->count_ == 0) delete p_; }
  FredPtr(const FredPtr& p) : p_(p.p_) { ++p_->count_; }
  FredPtr& operator= (const FredPtr& p)
        { // DO NOT CHANGE THE ORDER OF THESE STATEMENTS!
          // (This order properly handles self-assignment)
          // (This order also properly handles recursion, e.g., if a Fr
          Fred* const old = p_;
          p_ = p.p_;
          ++p_->count_;
          if (--old->count_ == 0) delete old;
          return *this;
        }
private:
  Fred* p_;     // p_ is never NULL
```

```
};
```

Naturally you can use nested classes to rename `FredPtr` to `Fred::Ptr`.

Note that you can soften the "never `NULL`" rule above with a little more checking in the constructor, copy constructor, assignment operator, and destructor. If you do that, you might as well put a `p_ != NULL` check into the "`*`" and "`->`" operators (at least as an `assert()`). I would recommend against an `operator Fred*()` method, since that would let people accidentally get at the `Fred*`.

One of the implicit constraints on `FredPtr` is that it must only point to `Fred` objects which have been allocated via `new`. If you want to be really safe, you can enforce this constraint by making all of `Fred`'s constructors `private`, and for each constructor have a `public` (`static`) `create()` method which allocates the `Fred` object via `new` and returns a `FredPtr` (not a `Fred*`). That way the *only* way anyone could create a `Fred` object would be to get a `FredPtr` ("`Fred* p = new Fred()`" would be replaced by "`FredPtr p = Fred::create()`"). Thus no one could accidentally subvert the reference counting mechanism.

For example, if `Fred` had a `Fred::Fred()` and a `Fred::Fred(int i, int j)`, the changes to `class Fred` would be:

```
class Fred {
public:
  static FredPtr create();              // Defined below class FredPtr
  static FredPtr create(int i, int j);  // Defined below class FredPtr
  // ...
private:
  Fred();
  Fred(int i, int j);
  // ...
};

class FredPtr { /* ... */ };

inline FredPtr Fred::create()             { return new Fred(); }
inline FredPtr Fred::create(int i, int j) { return new Fred(i,j); }
```

The end result is that you now have a way to use simple reference counting to provide "pointer semantics" for a given object. Users of your `Fred` `class` explicitly use `FredPtr` objects, which act more or less like `Fred*` pointers. The benefit is that users can make as many copies of their `FredPtr` "smart pointer" objects, and the pointed-to `Fred` object will automagically get `deleted` when the last such `FredPtr` object vanishes.

If you'd rather give your users "reference semantics" rather than "pointer semantics," you can use reference counting to provide "copy on write".

FAQ How do I provide reference counting with copy-on-write semantics?

Reference counting can be done with either pointer semantics or reference semantics. The previous FAQ shows how to do reference counting with pointer semantics. This FAQ shows how to do reference counting with reference semantics.

The basic idea is to allow users to think they're copying your `Fred` objects, but in reality the underlying implementation doesn't actually do any copying unless and until some user actually tries to modify the underlying `Fred` object.

Class `Fred::Data` houses all the data that would normally go into the `Fred` `class`. `Fred::Data` also has an extra data member, `count_`, to manage the reference counting. Class `Fred` ends up being a "smart reference" that (internally) points to a `Fred::Data`.

```
class Fred {
public:

  Fred();                                  // A default constructor
  Fred(int i, int j);                      // A normal constructor

  Fred(const Fred& f);
  Fred& operator= (const Fred& f);
 ~Fred();

  void sampleInspectorMethod() const;   // No changes to this object
  void sampleMutatorMethod();           // Change this object

  // ...
```

```
private:

  class Data {
  public:
    Data();
    Data(int i, int j);
    Data(const Data& d);

    // Since only Fred can access a Fred::Data object,
    // you can make Fred::Data's data public if you want.
    // But if that makes you uncomfortable, make the data private
    // and make Fred a friend class via friend class Fred;

    // ...your data members are declared here...

    unsigned count_;
    // count_ is the number of Fred objects that point at this
    // count_ must be initialized to 1 by all constructors
    // (it starts as 1 since it is pointed to by the Fred object that c
  };

  Data* data_;
};

Fred::Data::Data()              : count_(1) /*init other data*/ { }
Fred::Data::Data(int i, int j)  : count_(1) /*init other data*/ { }
Fred::Data::Data(const Data& d) : count_(1) /*init other data*/ { }

Fred::Fred()             : data_(new Data()) { }
Fred::Fred(int i, int j) : data_(new Data(i, j)) { }

Fred::Fred(const Fred& f)
  : data_(f.data_)
{
  ++data_->count_;
}

Fred& Fred::operator= (const Fred& f)
{
  // DO NOT CHANGE THE ORDER OF THESE STATEMENTS!
  // (This order properly handles self-assignment)
  // (This order also properly handles recursion, e.g., if a Fred::Data
  Data* const old = data_;
  data_ = f.data_;
  ++data_->count_;
  if (--old->count_ == 0) delete old;
  return *this;
}

Fred::~Fred()
{
  if (--data_->count_ == 0) delete data_;
}

void Fred::sampleInspectorMethod() const
{
  // This method promises ("const") not to change anything in *data_
  // Other than that, any data access would simply use "data_->..."
}

void Fred::sampleMutatorMethod()
{
  // This method might need to change things in *data_
  // Thus it first checks if this is the only pointer to *data_
  if (data_->count_ > 1) {
    Data* d = new Data(*data_);    // Invoke Fred::Data's copy ctor
    --data_->count_;
    data_ = d;
  }
  assert(data_->count_ == 1);
```

```
  // Now the method proceeds to access "data_->..." as normal
}
```

If it is fairly common to call `Fred`'s default constructor, you can avoid all those `new` calls by sharing a common `Fred::Data` object for all `Fred`s that are constructed via `Fred::Fred()`. To avoid `static` initialization order problems, this shared `Fred::Data` object is created "on first use" inside a function. Here are the changes that would be made to the above code (note that the shared `Fred::Data` object's destructor is never invoked; if that is a problem, either hope you don't have any `static` initialization order problems, or drop back to the approach described above):

```
class Fred {
public:
  // ...
private:
  // ...
  static Data* defaultData();
};

Fred::Fred()
  : data_(defaultData())
{
  ++data_->count_;
}

Fred::Data* Fred::defaultData()
{
  static Data* p = nullptr;
  if (p == nullptr) {
    p = new Data();
    ++p->count_;    // Make sure it never goes to zero
  }
  return p;
}
```

Note: You can also provide reference counting for a hierarchy of classes if your `Fred` class would normally have been a base class.

FAQ How do I provide reference counting with copy-on-write semantics for a hierarchy of classes?

The previous FAQ presented a reference counting scheme that provided users with reference semantics, but did so for a single class rather than for a hierarchy of classes. This FAQ extends the

previous technique to allow for a hierarchy of classes. The basic difference is that `Fred::Data` is now the root of a hierarchy of classes, which probably cause it to have some `virtual` functions. Note that class `Fred` itself will still not have any `virtual` functions.

The Virtual Constructor Idiom is used to make copies of the `Fred::Data` objects. To select which derived class to create, the sample code below uses the Named Constructor Idiom, but other techniques are possible (a `switch` statement in the constructor, etc). The sample code assumes two derived classes: `Der1` and `Der2`. Methods in the derived classes are unaware of the reference counting.

```
class Fred {
public:

  static Fred create1(const std::string& s, int i);
  static Fred create2(float x, float y);

  Fred(const Fred& f);
  Fred& operator= (const Fred& f);
 ~Fred();

  void sampleInspectorMethod() const;   // No changes to this object
  void sampleMutatorMethod();           // Change this object

  // ...

private:

  class Data {
  public:
    Data() : count_(1) { }
    Data(const Data& d) : count_(1) { }              // Do NOT copy the
    Data& operator= (const Data&) { return *this; }  // Do NOT copy the
    virtual ~Data() { assert(count_ == 0); }         // A virtual destr
    virtual Data* clone() const = 0;                 // A virtual const
    virtual void sampleInspectorMethod() const = 0;  // A pure virtual
    virtual void sampleMutatorMethod() = 0;
  private:
    unsigned count_;   // count_ doesn't need to be protected
    friend class Fred; // Allow Fred to access count_
  };

  class Der1 : public Data {
  public:
    Der1(const std::string& s, int i);
    virtual void sampleInspectorMethod() const;
    virtual void sampleMutatorMethod();
    virtual Data* clone() const;
    // ...
  };

  class Der2 : public Data {
  public:
    Der2(float x, float y);
    virtual void sampleInspectorMethod() const;
```

```
    virtual void sampleMutatorMethod();
    virtual Data* clone() const;
    // ...
  };

  Fred(Data* data);
  // Creates a Fred smart-reference that owns *data
  // It is private to force users to use a createXXX() method
  // Requirement: data must not be NULL

  Data* data_;   // Invariant: data_ is never NULL
};

Fred::Fred(Data* data) : data_(data)  { assert(data != nullptr); }

Fred Fred::create1(const std::string& s, int i) { return Fred(new Der1(
Fred Fred::create2(float x, float y)            { return Fred(new Der2(

Fred::Data* Fred::Der1::clone() const { return new Der1(*this); }
Fred::Data* Fred::Der2::clone() const { return new Der2(*this); }

Fred::Fred(const Fred& f)
  : data_(f.data_)
{
  ++data_->count_;
}

Fred& Fred::operator= (const Fred& f)
{
  // DO NOT CHANGE THE ORDER OF THESE STATEMENTS!
  // (This order properly handles self-assignment)
  // (This order also properly handles recursion, e.g., if a Fred::Data
  Data* const old = data_;
  data_ = f.data_;
  ++data_->count_;
  if (--old->count_ == 0) delete old;
  return *this;
}

Fred::~Fred()
{
  if (--data_->count_ == 0) delete data_;
}

void Fred::sampleInspectorMethod() const
{
  // This method promises ("const") not to change anything in *data_
  // Therefore we simply "pass the method through" to *data_:
  data_->sampleInspectorMethod();
}

void Fred::sampleMutatorMethod()
{
  // This method might need to change things in *data_
  // Thus it first checks if this is the only pointer to *data_
  if (data_->count_ > 1) {
    Data* d = data_->clone();   // The Virtual Constructor Idiom
    --data_->count_;
    data_ = d;
  }
  assert(data_->count_ == 1);

  // Now we "pass the method through" to *data_:
  data_->sampleMutatorMethod();
}
```

Naturally the constructors and `sampleXXX` methods for `Fred::Der1` and `Fred::Der2` will need to be implemented in whatever way is appropriate.

FAQ Can I absolutely *prevent* people from subverting the reference counting mechanism, and if so, *should* I?

No, and (normally) no.

There are two basic approaches to subverting the reference counting mechanism:

1. The scheme could be subverted if someone got a `Fred*` (rather than being forced to use a `FredPtr`). Someone could get a `Fred*` if class `FredPtr` has an `operator*()` that returns a `Fred&`: `FredPtr p = Fred::create(); Fred* p2 = &*p;`. Yes it's bizarre and unexpected, but it could happen. This hole could be closed in two ways: overload `Fred::operator&()` so it returns a `FredPtr`, or change the return type of `FredPtr::operator*()` so it returns a `FredRef` (`FredRef` would be a class that simulates a reference; it would need to have all the methods that `Fred` has, and it would need to forward all those method calls to the underlying `Fred` object; there might be a performance penalty for this second choice depending on how good the compiler is at inlining methods). Another way to fix this is to eliminate `FredPtr::operator*()` — and lose the corresponding ability to get and use a `Fred&`. But even if you did all this, someone could still generate a `Fred*` by explicitly calling `operator->()`: `FredPtr p = Fred::create(); Fred* p2 = p.operator->();`.
2. The scheme could be subverted if someone had a leak and/or dangling pointer to a `FredPtr`. Basically what we're saying here is that `Fred` is now safe, but we somehow want to prevent people from doing stupid things with `FredPtr` objects. (And if we could solve that via `FredPtrPtr` objects, we'd have the same problem

again with them). One hole here is if someone created a `FredPtr` using `new`, then allowed the `FredPtr` to leak (worst case this is a leak, which is bad but is *usually* a little better than a dangling pointer). This hole could be plugged by declaring `FredPtr::operator new()` as `private`, thus preventing someone from saying `new FredPtr()`. Another hole here is if someone creates a local `FredPtr` object, then takes the address of that `FredPtr` and passed around the `FredPtr*`. If that `FredPtr*` lived longer than the `FredPtr`, you could have a dangling pointer — shudder. This hole could be plugged by preventing people from taking the address of a `FredPtr` (by overloading `FredPtr::operator&()` as `private`), with the corresponding loss of functionality. But even if you did all that, they could still create a `FredPtr&` which is almost as dangerous as a `FredPtr*`, simply by doing this: `FredPtr p; ... FredPtr& q = p;` (or by passing the `FredPtr&` to someone else).

And even if we closed *all* those holes, C++ has those wonderful pieces of syntax called pointer casts. Using a pointer cast or two, a sufficiently motivated programmer can normally create a hole that's big enough to drive a proverbial truck through. (By the way, pointer casts are evil.)

So the lessons here seem to be: (a) you can't prevent espionage no matter how hard you try, and (b) you can easily prevent mistakes.

So I recommend settling for the "low hanging fruit": use the easy-to-build and easy-to-use mechanisms that prevent mistakes, and don't bother trying to prevent espionage. You won't succeed, and even if you do, it'll (probably) cost you more than it's worth.

So if we can't use the C++ language itself to prevent espionage, are there other ways to do it? Yes. I personally use old fashioned code reviews for that. And since the espionage techniques usually involve some bizarre syntax and/or use of pointer-casts and unions, you can use a tool to point out most of the "hot spots."

FAQ Can I use a garbage collector in C++?

Yes.

If you want automatic garbage collection, there are good commercial and public-domain garbage collectors for C++. For applications where garbage collection is suitable, C++ is an excellent garbage collected language with a performance that compares favorably with other garbage collected languages. See The C++ Programming Language (4th Edition) for a discussion of automatic garbage collection in C++. See also, Hans-J. Boehm's site for C and C++ garbage collection .

Also, C++ supports programming techniques that allows memory management to be safe and implicit without a garbage collector. Garbage collection is useful for specific needs, such as inside the implementation of lock-free data structures to avoid ABA issues, but not as a general-purpose default way of handling for resource management. We are not saying that GC is not useful, just that there are better approaches in many situations.

C++11 offers a GC ABI.

Compared with the "smart pointer" techniques, the two kinds of garbage collector techniques are:

- less portable
- usually more efficient (especially when the average object size is small or in multithreaded environments)
- able to handle "cycles" in the data (reference counting techniques normally "leak" if the data structures can form a cycle)
- sometimes leak other objects (since the garbage collectors are necessarily conservative, they sometimes see a random bit pattern that appears to be a pointer into an allocation, especially if the allocation is large; this can allow the allocation to leak)
- work better with existing libraries (since smart pointers need to be used explicitly, they may be hard to integrate with existing libraries)

FAQ What are the two kinds of garbage collectors for C++?

In general, there seem to be two flavors of garbage collectors for C++:

1. *Conservative garbage collectors.* These know little or nothing about the layout of the stack or of C++ objects, and simply look for bit patterns that appear to be pointers. In practice they seem to work with both C and C++ code, particularly when the average object size is small. Here are some examples, in alphabetical order:
   - Boehm-Demers-Weiser collector
   - Geodesic Systems collector
2. *Hybrid garbage collectors.* These usually scan the stack conservatively, but require the programmer to supply layout information for heap objects. This requires more work on the programmer's part, but may result in improved performance. Here are some examples, in alphabetical order:
   - Attardi and Flagella's CMM
   - Bartlett's mostly copying collector

Since garbage collectors for C++ are normally conservative, they can sometimes leak if a bit pattern "looks like" it might be a pointer to an otherwise unused block. Also they sometimes get confused when pointers to a block actually point outside the block's extent (which is illegal, but some programmers simply *must* push the envelope; sigh) and (rarely) when a pointer is hidden by a compiler optimization. In practice these problems are not usually serious, however providing the collector with hints about the layout of the objects can sometimes ameliorate these issues.

FAQ Where can I get more info on garbage collectors for C++?

For more information, see the Garbage Collector FAQ .

FAQ What is an `auto_ptr` and why isn't there an `auto_array`?

It's now spelled `unique_ptr`, which supports both single objects and arrays.

`auto_ptr` is an old standard smart pointer that has been deprecated, and is only being kept in the standard for backward compatibility with older code. It should not be used in new code.